**FUNDAMENTAL PROGRAMMING TECHNIQUES**

**ASSIGNMENT 3**

**ORDER MANAGEMENT**

Anghel Dan-Marian

Group 30422

Professor: Diana Balc

**Table of contents:**

1. **Objectives**
2. **Analysis**
3. **Development**
4. **Implementation**
5. **Results**
6. **Conclusion**
7. **Bibliography**

**1. Objectives**

**Primary objectives:**

Consider an application Order Management for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application should be structured in packages using a layered architecture and should use (minimally) the following classes:

• Model classes - the data models of the application

• Business Logic classes – implement the application logic

• Presentation classes – implement the user input/output

• Data access classes - implement the access to the database

Use Javadoc for documenting the classes and their functionality. Create a jar file in order for the application to be more easily executed. Generate PDF files to capture information required by certain operation. Create a SQL dump file containing the SQL statements for creating the database and the tables and populating the tables with the corresponding data.

The application should allow processing commands from a text file given as argument, perform the requested operations, save the data in the database, and generate reports in pdf format. Other classes and packages can be added to implement the full functionality of the application. Implement a parser to read commands in the Presentation layer (instead of the standard graphical user interface), and a pdf file generator to generate reports, according to the following examples:

|  |  |  |
| --- | --- | --- |
| Command name | Command Syntax | Description |
| Add client to the database | Insert client: Ion Popescu, Bucuresti | Insert in the database a new client with name Ion Popescu and address Bucuresti |
| Delete Client from the database | Delete client: Ion Popescu | Delete from database the client with name Ion Popescu |
| Add product to the database | Insert product: apple, 20, 1 | Add product apple with quantity 20 and price 1 |
| Delete product from the database | Delete product: apple | Delete product apple from database |
| Create order for client | Order: Ion Popescu, apple, 5 | Create order for Ion Popescu, with apple quantity 5. Also update the apple stock to 15. Generate a bill in pdf format with the order and total price of 5 |
| Generate reports | Report client  Report order  Report product | Generate pdf reports with all clients/orders/products displayed in a tabular form. The reports should contain the information corresponding to the entity for which reports are asked (client, order or product) returned from the database by a SELECT \* query, displayed in a table in a PDF file |

**Secondary objectives:**

* **Use-cases** (Chapter 2):  a **use case** is a list of actions or event steps typically defining the interactions between a role and a system to achieve a goal, usually represented trough a flow-chart or an UML diagram.
* **Diagrams** (Chapter 3): they provide a mock-up for the project so that the developer can more easily grasp the concept and the small details of it.
* **Data structures** (Chapter 3): the types of data structures used in making the application.
* **The classes used** (Chapter 3): the project has been split in smaller classes that can accomplish certain tasks and provide relevant results through reflection.
* **Packages** (Chapter 3): the project is split in multiple packages in order to accomplish a layered architecture based on the model: data access layer, business layer, model and presentation.
* **Implementation** (Chapter 4): each class will be explained for a better understanding of the functioning of the program.
* **Results** (Chapter 5): the results are given in the form of PDF files for bills and reports and the SQL dump file, but will be talked over again here nevertheless.

**2. Problem Analysis**

* **General Overview**

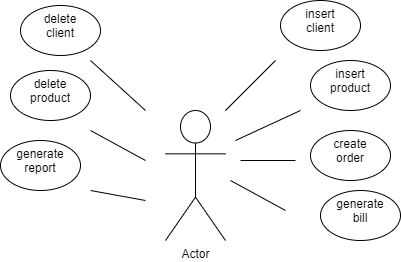
This application should be able to fulfil all the requirements in order to display, modify, and keep track of orders, clients and products. These are stored in a relational MySQL database, along with the information about the users which have access to the system. This way, all the data is easier to retrieve and access from different computers.

* **Input and output**

The input is given by the user in the form of a text file containing the commands and the arguments that need to be processed by the application. The user has the option to add a client or a product, create an order for a product, generate a bill from an order, delete a client or a product and finally, generate a report for a table of his/hers choosing.

The output is generated in the form of PDF files or a SQL dump file. The PDF files are in the form of bills, which give information about the client that placed the order, the product that was ordered and the amount to be paid, and in the form of reports which give information about a table as a whole.

* **Use cases**



(\* Use-case diagram of the program)

* **Insert Client**

|  |  |
| --- | --- |
| Use Case | Insert client |
| Actor | End-User |
| Basic Flow | This can be accomplished by the user through a command given inside the input text file. The user will specify the name of the client and the city. |
| Alternate Flow | In case the client already exists or an error occurred during the creation, the user will be notified with a message inside the terminal that the program was ran inside. |

* **Insert Product**

|  |  |
| --- | --- |
| Use Case | Insert product |
| Actor | End-User |
| Basic Flow | This can be accomplished by the user through a command given inside the input text file. The user will specify the name of the product, the price and the stock (which can be added instead, if the product already exists). |
| Alternate Flow | In case that an error occurred during the creation, the user will be notified with a message inside the terminal that the program was ran inside. If the product already exists the specified stock will be added to the existing one instead, without having to notify the user. |

* **Create Order**

|  |  |
| --- | --- |
| Use Case | Create order |
| Actor | End-User |
| Basic Flow | This can be accomplished by the user through a command given inside the input text file. The user will specify the name of the client that placed the order, the product that is ordered and the amount. |
| Alternate Flow | In case that an error occurred during the creation, the user will be notified with a message inside the terminal that the program was ran inside. |

* **Generate Bill**

|  |  |
| --- | --- |
| Use Case | Generate bill |
| Actor | End-User |
| Basic Flow | This action will be automatically performed by the program whenever an order is placed. The bill is generated inside a PDF file and it gives information about the client that placed the order, the product that was ordered and the total amount to be paid. |
| Alternate Flow | In case that an error occurred during the creation of the file, the user will be notified with a message inside the terminal that the program was ran inside. |

* **Delete Client**

|  |  |
| --- | --- |
| Use Case | Delete client |
| Actor | End-User |
| Basic Flow | This can be accomplished by the user through a command given inside the input text file. The user will specify the name of the client and the program will search the data base to see if the client exists. If he/she exists then the command is complete and the client is deleted. |
| Alternate Flow | In case that the client that we want to delete does not exist or an error occurred during the execution of the program, the user will be notified with a message inside the terminal that the program was ran inside. |

* **Delete Product**

|  |  |
| --- | --- |
| Use Case | Delete product |
| Actor | End-User |
| Basic Flow | This can be accomplished by the user through a command given inside the input text file. The user will specify the name of the product and the program will search the data base to see if the product exists. If it exists then the command is complete and the product is deleted. |
| Alternate Flow | In case that the product that we want to delete does not exist or an error occurred during the execution of the program, the user will be notified with a message inside the terminal that the program was ran inside. |

* **Generate Report**

|  |  |
| --- | --- |
| Use Case | Generate report |
| Actor | End-User |
| Basic Flow | This can be accomplished by the user through a command given inside the input text file. The user will specify the name of the table that they want to generate the report for. The report will be generated as a PDF file containing the table as a whole. |
| Alternate Flow | In case that an error occurred during the execution of the program, the user will be notified with a message inside the terminal that the program was ran inside. |

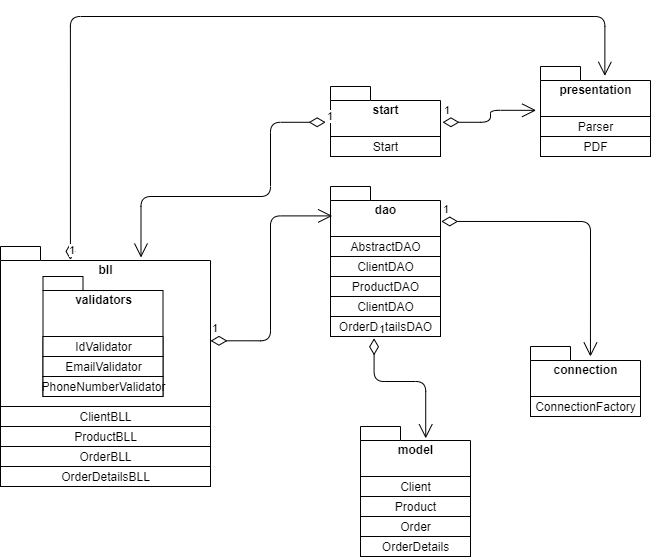
**3. Development**

The project follows layered structure specific to an application based on relational data bases, i.e. data access layer, business layer, model and presentation. The usual graphic interface that can be found in the presentation layer is replaced by 2 classes that are tasked with parsing the input data and creating output pdf files which are easy to interpret by the user.

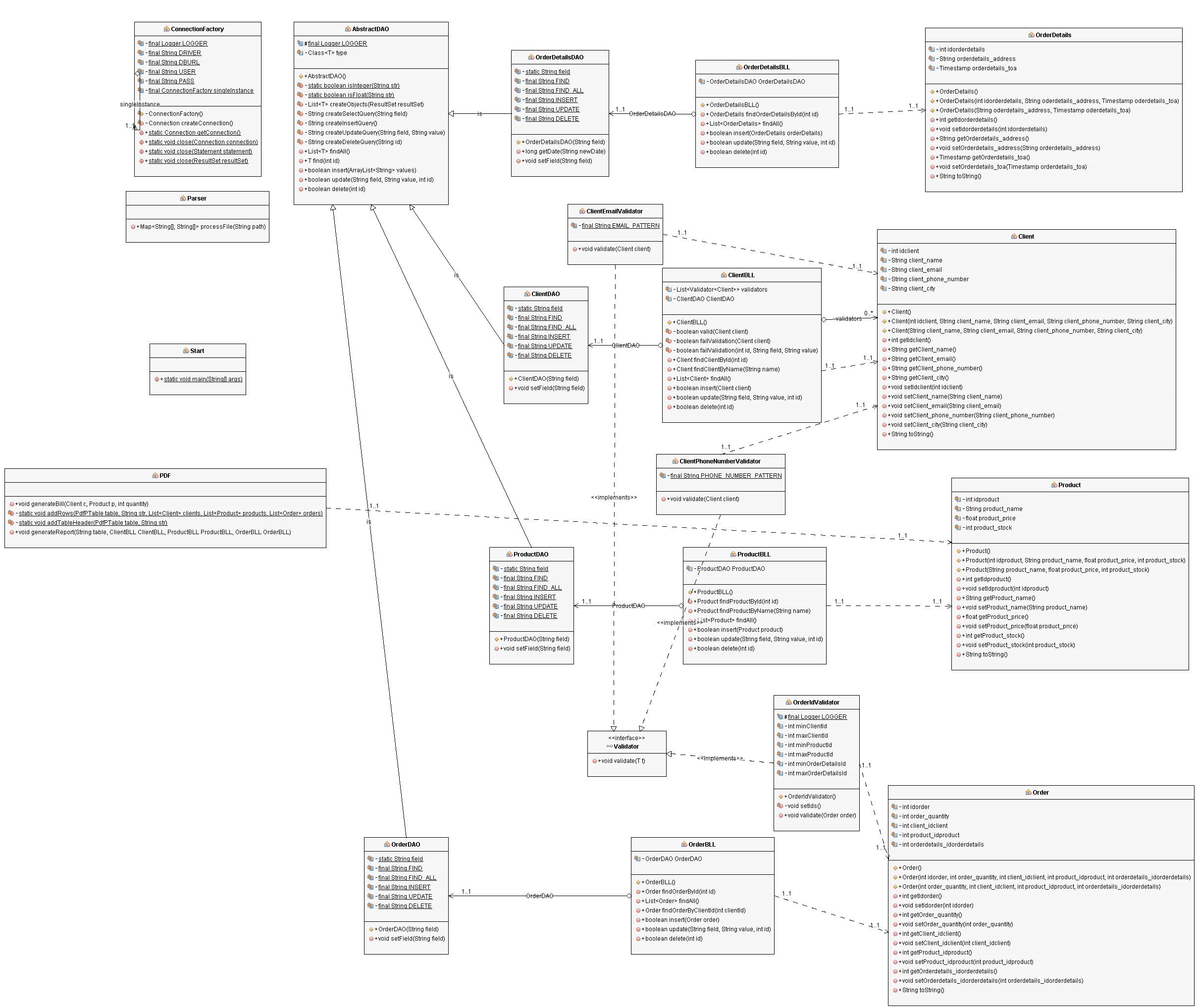
The classes are structured such that they can be implemented by reflection techniques, i.e. to create a generic class that contains the methods for accessing the DB: create object, edit object, delete object and find object. The queries for accessing the DB for a specific object that corresponds to a table will be generated dynamically through reflection.

Other than the usual lists, the data structure of choice for this project was the linked hash map which was used to store the commands given by the user inside then input text file. Due to the form of the commands (like a function and parameters) a map is optimal to store the operations as the keys and the parameters as the values.

This is the package diagram of the project where it can be observed the layered structure (data access layer, business layer, model and presentation).

(\* Package diagram of the project)

This is the class diagram of the project, where it can be observed the MVC, monomial, polynomial structure described above.



(\* Class diagram of the project)

This image has been generated using the easyUML plugin of the NetBeans IDE.

A better-quality representation can be observed at the link: <https://imgur.com/a/Xxcfl48>

**4. Implementation**

(\* describing each package and their classes)

**4.1 start**

* **Start** – This is the main class of the project, it is responsible to take the processed data from the parser object and decide which operation needs to be performed and which parameters need to be used.

It uses an object of type PDF in order to generate bills and reports whenever that command is issued.

Furthermore, it contains one of each object of the type bll (client, product, order, order details), which, trough the dao type object, pull the data from the data base (or insert it).

**4.2 presentation**

* **Parser** – This class takes the input text file and processes it into a linked hash map that contains the operations (ex. Insert client) as the keys, and the parameters (ex. Ion Pop, Bucuresti) as the values.
* **PDF** – This class is meant to generate PDF files as bills and reports. For this we a library called iText (check bibliography).

The bill is very straight forward, it just takes the information about

the client that placed the order, the information about the product that was ordered, the total cost of the order and places the in a PDF file.

The report is a little more complex due to the fact that the information needs to be exported in a tabular form. For this we use a function that creates the header of the table (i.e. the first row) and one that fills in with the rest of the cells depending on the selected table of the data base.

**4.3 model**

* **Client** – This class is the model class for the “client” data base table. It contains the same fields as the table, a few constructors, the overridden to string method and the corresponding setters and getters.
* **Product** – This class is the model class for the “product” data base table. It contains the same fields as the table, a few constructors, the overridden to string method and the corresponding setters and getters.
* **Order** – This class is the model class for the “order” data base table. It contains the same fields as the table, a few constructors, the overridden to string method and the corresponding setters and getters.

**4.4 dao**

* **AbstractDAO** – This class provides access to the information of the data base through methods built with the reflection technique. It has methods for finding, inserting, updating or deleting an object.
* **ClientDAO** – This class “extends” the previous mentioned one by giving it the type “client” and provides a few client specific queries.
* **ProductDAO** – This class “extends” the previous mentioned one by giving it the type “product” and provides a few product specific queries.
* **OrderDAO** – This class “extends” the previous mentioned one by giving it the type “order” and provides a few order specific queries.

**4.5 bll**

* **ClientBLL** – This class uses an object of type “dao” to pull/send information from/to the data base. It implements the methods from the general class made by reflection and transforms them to be suitable to the “client” table.
* **ProductBLL** – This class uses an object of type “dao” to pull/send information from/to the data base. It implements the methods from the general class made by reflection and transforms them to be suitable to the “product” table.
* **OrderBLL** – This class uses an object of type “dao” to pull/send information from/to the data base. It implements the methods from the general class made by reflection and transforms them to be suitable to the “order” table.

**5. Results**

The results are given in the form of PDF files or a SQL dump file. The PDF files are in the form of bills, which give information about the client that placed the order, the product that was ordered and the amount to be paid, and in the form of reports which give information about a table as a whole.

**6. Conclusions**

The order management project was a great remembering exercise to strengthen the knowledge acquired last semester during the data base classes/laboratories. Furthermore, this project managed to get me involved in learning new and exciting programming techniques, as well as some nice tricks. The reflection technique is a very powerful tool in the arsenal of any programmer, such that it allows high levels of abstraction and automation.

There is great improvement potential to this project in the form of adding new operations to retrieve and manipulate the database. There is even room to add some different type of functionality like, sorting the data or giving the user a graphical interface for an easier way of interacting with the application or even give the option to interact with multiple data bases.

**7. Bibliography**

* Class diagram of the project: <https://imgur.com/a/Xxcfl48>
* EasyUML plugin for generating class diagrams inside NetBeans: <http://plugins.netbeans.org/plugin/55435/easyuml>
* Easy to use website to generate UML diagrams, flowcharts, etc.: <https://app.diagrams.net/>
* iText library <https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22itextpdf%22>